6. Knowledge Modeling

From **“**Verification, Validation, and Evaluation of Expert Systems, Volume I”

This chapter presents some *knowledge models* that can be used to partition knowledge bases using expert knowledge. The chapter includes:

- Definition of knowledge models.
- Using knowledge models for VV&E.
- Using knowledge models in the expert system lifecycles.
- Some example knowledge models.
- Proof techniques for specific knowledge models.
- Specific knowledge models

Appendix A presents some mathematical results used in the chapter about partitioning using the clear box methodology.

Introduction

Knowledge models are high level templates for expert knowledge. Examples of knowledge models are decision trees, flowcharts and state diagrams. By organizing the knowledge, a knowledge model helps with VV&E by suggesting strategies for proofs and partitions; in addition, some knowledge models have mathematical properties that help establish completeness, consistency or specification satisfaction.

More particularly:

- The knowledge model highlights the main points of a knowledge base, often obscured in the knowledge base.
- A knowledge model partitions a large KB into smaller, easier to verify, pieces.
- There are mathematical properties of the knowledge model that help establish the correctness of a knowledge base.

An Example of a Knowledge Model

PAMEX (Pavement Maintenance Expert System) is an expert system for pavement maintenance management [Aougab et. al., 1988]. A top level model of PAMEX consists of a partition of the problem space on the following three variables:

- Level of information about the pavement; the 3 values are extensive, some and little or none.
- Range of pavement serviceability index (PSI); the 3 values are above 2.8, between 2.8 and 2.0, and below 2.0.
- The level of treatment desired; the 3 values are long-range, mid-term and short-term.

For each of the twenty seven regions formed by the Cartesian product of the three regions on each variable, there is a small expert system that handles problems in that region. These small expert systems use the same pavement variables, i.e., PSI and other more specific pavement measurements. In this case, the model is a decision tree, discussed and illustrated in the next section.

Using Knowledge Models in VV&E

The steps in using a knowledge model in VV&E are:

- Collect the knowledge model from:
  - The domain expert(s) working on the project.
  - Standards documents in the domain.
  - Notes from knowledge acquisition at the time an existing system was built.
- Validate the knowledge; see Chapter 9 on knowledge validation for details. This step is to ensure that the knowledge going into the expert system represents correct expert knowledge.
- Prove the expert system using the knowledge model is complete, consistent and satisfies its specifications; this chapter, as well as chapters on partitioning and small systems, provides information on how to develop these proofs.

Decision Trees

Introduction

A decision tree is a set of decisions that partitions the input space into a set of disjoint regions that cover the entire input space. In a decision tree system, a sequence of decisions based on user input and other data are used to classify the input problem before going on to the rest of problem solution.

The top of the decision tree corresponds to the start of the decision process. At each interior node of a decision tree, the problem is supposed to be assigned to one and only one of the subnodes. The solution of the detailed problems is often handled by specialized expert systems tailored to the specialized situations found by the decision tree.

Definition

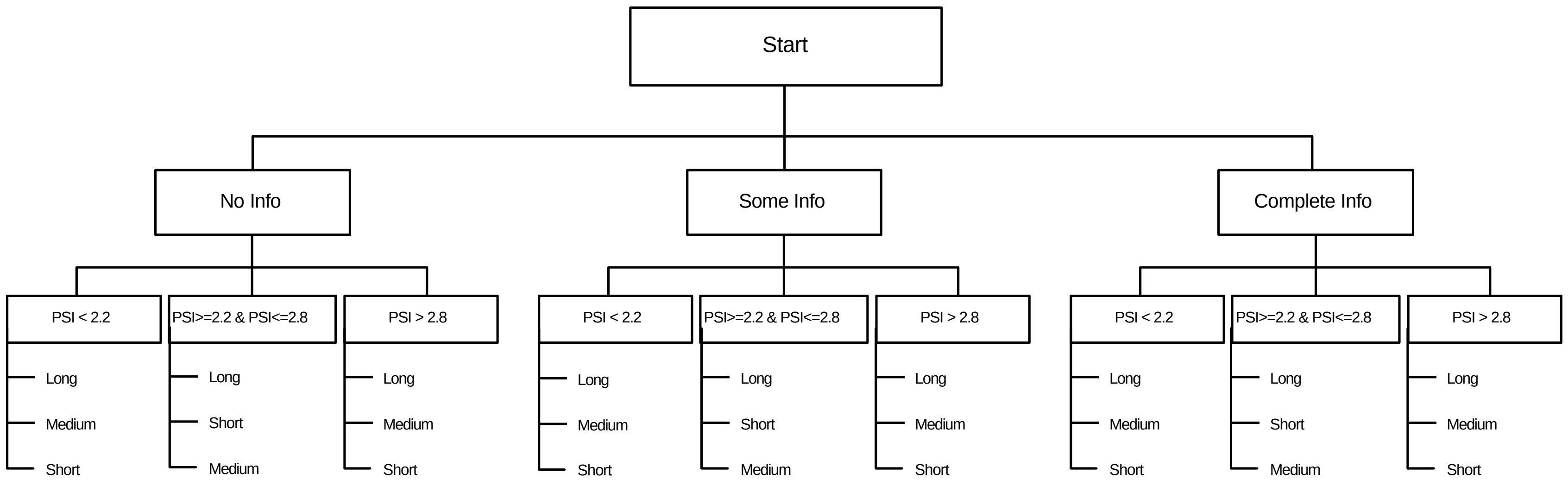
A decision tree expert system has a structure that is described by a tree. A decision tree system has the following properties:

- Each interior node of the tree has a variable or expression assigned to it.
- Each edge to a subtree is labeled with a set of values for that variable or expression on the parent node.
- All possible values of a variable are on some edge.
- No variable value is on two different sibling edges.
- Associated with each leaf node is a subsystem or output(s). A subsystem at a tip node N of a decision tree is called to solve the problems for which variables appearing in the tree have values associated with the path that leads to N.

Example

A decision tree for PAMEX is illustrated in figure 7.1 of the following page.

PAMEX Decision Tree



LEGEND

PSI: Pavement Serviceability Index

Info: the amount of information available about the pavement

short, medium, long term: the time period for which the fix is made, subject to budget constraints

Pavement Maintenance Expert System

Figure 6.1: Pamex DT

Use During Development

Decision trees are a useful way to organize expert knowledge. Their use is indicated when the expert can describe in what order information is obtained and used to partially determine a solution. Drawing a decision tree from information the expert(s) have provided is a good way to present the knowledge engineer's conception of the information back to the domain expert for validation.

Use During VV&E

To model an expert system as a decision tree for the purpose of showing correctness, the following conditions should be satisfied:

- Each possible set of inputs should be in one and only one of the partitions generated by the decision tree.
- For each partition, there is an expert system (a subsystem of the entire system) that correctly solves problems in that partition.
- Experts validate the decision tree.
- The expert system assigns each input to the correct partition as the result of a finite computation.

To prove *completeness* of an expert system modeled by a decision tree, prove the following:

- Each possible problem in the input space is assigned to some partition of the decision tree.
- Each expert system assigned to one of the partitions computes a solution for each problem assigned to it.

To prove *consistency* of an expert system modeled by a decision tree, prove the following:

- Each possible problem in the input space is assigned to at most one partition of the decision tree.
- Each expert system assigned to one of the partitions computes at most one solution for each problem assigned to it.
- Each computed solution is internally consistent.

To prove *satisfaction of a requirement* of an expert system modeled by a decision tree, it needs to be shown that the requirement is satisfied for the expert system associated with each tip of the decision tree.

Ripple Down Rules

Introduction

Ripple down rules (RDR’s) [Kang, et al, 1994.] are a special case of decision trees for reasoning with defaults. RDR’s are guaranteed to be complete and consistent.

Definition

With ripple down rules, the knowledge base is organized as lists of rules. If the conditions ("if" part) of a rule are satisfied, then the expert system moves to the part of the knowledge base attached to this rule. In some cases, this is another list of rules. If so, the expert system tests the rules in the sublist. If there is no sublist of rules, or if none of the sublist rules are satisfied, then the conclusions "then part" of the rule is used. Figure 6.2 demonstrate an example of a small expert system for vehicles classification.

Example

As an example, a small expert system for vehicles classification is presented.

The main list is:

L1.1: If NOA (Number -of Axles) is 2
  Try List 1-1; Default = Car.
L1.2: If NOA is 3,
  Try List 1-2; Default = 3 Axle-single unit Truck.
L1.3: If NOA is 4,
  Try List 1-3; Default = 4 Axle-single unit Truck.
L1.4: If NOA is 5,
  Try List 1-4; Default = 5 Axle-single unit Truck.
etc.

Here are the lists that fill out the next level of the knowledge base; note that this is not an exhaustive knowledge base.

L1-1.1: If S1 <= 12, it is a Car-Van-Pick up.
L1-1.2: If S1 <= 20, it is a 2 Axle-single unit Truck.
L1-1.3: If S1 > 20, it is a 2 Axle Bus.

L1-2.1: If S1 <= 12 & 8 < S2 <= 18, it is a Light Vehicle w/ Single Axle Trailer.
L1-2.2: If 7 < S1 <= 20 & S2 <= 8, it is a 3 Axle-singular unit Truck.
L1-2.3: If S1 > 20 & S2 <= 8, it is 3 Axle Bus.
L1-2.4: If Else, it is a 2 Axle Tractor w/ Singular Axle Trailer.

L1-3.1: If S1 > 7 & S2 + S3 <= 12, it is a 4 Axle-singular unit Truck.
L1-3.2: If S1 > 7 & S2 <= 8 & S3 > 6, it is a 3 AxleTractor w/ Singular Axle Trailer.
L1-3.3: If Else, it is a 2 Axle Tractor w/ Tandem Axle Trailer.

L1-4.1: If S2+S3+S4 < 16, it is a 5 Axle-singular unit Truck.
L1-4.2: If S2 <= 8 & S4 <= 10.5, it is a 3 Axle Tractor w/ Tandem Axle Trailer.

L1-4.3: If S2 > 8 & S3 + S4 <= 12, it is a 2 Axle Tractor w/ Tridem Axle Trailer.
L1-4.4: If S2 > 8 & 12 < S3 + S4 <= 16, it is a 2 Axle Tractor w/ Tridem Axle Trailer Split.
etc.

Figure 6.2: Example ES (continued)

Similar rule lists could expand lists 1-3 and 1-4.

The expert system starts the example and the system moves to list 1-2 (likewise for the other L1 rules). If none of the entry conditions to the rules in list L1 is satisfied, the default of L1, car, is the KB conclusion.

Under the condition that NOA is 3, the system moves to list 1-2 and if none of the entry conditions to those rules is satisfied, the default of L2, axle-single unit truck, is the KB conclusion.

Use During Development

Kang et al., 1994 point out that it is possible to add correction rules to a running ripple down rules expert system. Whenever an error occurs, that error gets added to the last list of rules which the system tried before choosing an erroneous default.

Ripple down rule systems are ideally suited to problems where knowledge has the following structure:

- Early decisions made on a problem narrow the range of possible solutions, while later decisions pick particular solutions from a selected class.
- There is a default solution at each stage of the solution process.

Changing a Ripple Down Rule System

Ripple down rules are a special type of decision tree. For a knowledge base that consists of a series of more detailed decisions, but where the bases of the more detailed decisions vary for different points of the decision tree, the ripple down rules model is appropriate.

Given: an RDR, and a rule (if C then A) which the algorithm should execute, the algorithm *change* modifies the KB to make (if C then A) part of the system:

case 1: Top level list of RDR is empty.

If default(RDR) = A, do nothing,

else insert (if C then A) as a 1-element list of RDR.

case 2: The conditions on the first rule in the top level list of RDR = C.

Attach to the first rule the RDR with default = A and empty rule list.

case 3: The conditions on the first rule subsume C.

Replace the RDR attached to the first rule, denoted by R2, with *change*(R2).

case 4: C subsumes the conditions on the first rule.

Replace the first rule with (if C then A).

case 5: C and the conditions of the first rule can be simultaneously satisfied.

Insert (if C then A) before the first rule.

otherwise: Let RDR = H++T, where H is the first rule in the top level list, and T is the rest of the rules. Insert (if C then A) in T.

Use During VV&E

Completeness of a RDR system follows from the following theorem:

A Ripple-Down-Rule System is Complete.

Proof: Note that part of an RDR system attached to a top level rule is itself an RDR system.

Define the level of an RDR system as follows: If the system has only 1 rule list, it is of level 1. If the system has N+1 rule lists, then it has level 1+Max(level of RDR subsystems of the top level rule list).

Let R be an RDR system of level N+1. Assume all RDR systems of level N are complete. For any input, either some top level condition is satisfied or not. In the latter case, the system concludes the default. In the former case, the system finds the conclusion computed by RDR rules from the first satisfied top level rule. If there is a rule list associated with that condition, the conclusion is from an RDR system of level at most N, and so exists. If there is no rule list, the conclusion is from the condition itself. Therefore an RDR system produces a conclusion in all cases.

In a similar way, it can be proven that all RDR systems are *consistent*. Consistency, however, requires an additional check: that the conclusions associated with each path through the ripple down rule tree are consistent.

Satisfaction Of Specification: To verify that an RDR satisfies a proposition P:

1. Verify or modify the default of the top level rule set.

2. Verify or modify the first rule, if any in the top level list to satisfy P.

3. Verify or modify the RDR system attached to the first rule, if any.

4. Let RDR = H++T, where H is the first rule in the top level list, and T is the rest of the rules. Verify or modify T to satisfy P.

Generalizations Of RDR: A generalization of RDR systems occurs when the conditions in RDR rules are replaced with specialized expert systems whose purpose is to make the decision specified in the if part of the RDR rule. When, in an ordinary RDR system an RDR rule if part is evaluated, a generalized RDR system may call an expert subsystem. This is a backward chaining process although RDR systems are more structured than general backward chaining systems.

The same algorithms for VV&E on RDR systems also work for generalized systems, provided that the expert subsystems carry out the tests provided in the rule condition that the subsystem replaces.

State Diagrams

Introduction

A state diagram is a useful formal representation for the top level of process control expert systems.

Definition

A state diagram system is one where there is a unique *state* at every step of a solution, and at each state, there is a function that determines the next state.

Example

A state diagram can be used to model driver behavior on a road segment. A set of *states* indicates the situation and/or goal of the driver. For example, some possible states are:

- Distance ahead too small.
- Clear road ahead.
- Approaching desired exit.

A driver model based on these states is shown below. The case statement branches on the value of the variable *state*.

```
state = start_loop;
while ( state is not equal to exit )
case (state)
[
 case start_loop:
            if (distance ahead is too small)
                        state = distance ahead too small;
            else (approaching desired exit)
                        state = exit;
            else (clear road ahead)
                        state = clear road ahead;
            else delay a small time increment;
 case clear road ahead:
            if (current speed < desired speed)
                        increment speed;
            delay a small time increment;
```

```
          state = start_loop;
 case distance ahead too small:
          if (current speed < desired speed)
                   { if (passing possible)
                            pass;
                     else decrease speed; }
          delay a small time increment;
          state = start_loop;
 case exit;
          return any current useful information to calling program
}
```

In this example, the decision to pass may be made by another expert system. In addition, fuzzy logic is often used to assign a membership grade representing how much the current situation belongs to each of the possible states. In this case, the expert system chooses a state with the highest membership grade and executes the code associated with that state.

State Diagram Systems Represented as Rules: Systems based on state diagrams may be encoded into expert system rules. The following include two of the rules that would implement the above example in rule form:

```
if state = start_loop
          and distance ahead is too small
          then state = distance ahead too small.
if state = start_loop
          and approaching desired exit
          then exit and return information to calling program
if state = start_loop
          and clear road ahead
          then state = clear road ahead
if state = start_loop
          and not ( distance ahead is too small
                   or approaching desired exit
                   or clear road ahead)
          then delay a small time increment
if state = clear road ahead
          and current speed < desired speed
          then increment speed
                   and delay a small time increment
                   and state = start_loop;
if state = clear road ahead
          and current speed >= desired speed
```

```
        then and state = start_loop;
if state = distance ahead too small
        and current speed < desired speed
        and passing possible
        then pass
                and delay a small time increment
                and state = start_loop
if state = distance ahead too small
        and current speed < desired speed
        and not passing possible
        then decrease speed
                and delay a small time increment
                and state = start_loop
if state = distance ahead too small
        and current speed >= desired speed
        then decrease speed
                and delay a small time increment
                and state = start_loop
```

Use During Development

State diagram models are useful during development when expert knowledge has the following characteristics:

- The problem solution consists of a series of distinct steps.
- Which step to choose is a complex, but knowledge-based decision.
- The possible paths through the steps may contain loops.

To run such a rule-based system based on state diagrams generally requires an inference engine that can do both forward and backward chaining with the same knowledge base in a strategy called forward chaining with local backward chaining. In this strategy applied to the knowledge base forward chaining keeps applying rules until a rule containing the command to exit the knowledge base fires. Backward chaining is used to establish the conditions within the rules, e.g., passing possible in the above example.

Use During VV&E

Completeness of a state diagram system can be established by showing that for any inputs the system eventually reaches a *final* state where it returns information and exits to the calling environment. In a complex system in which the predicates that control transitions between states are themselves expert systems, the proof of completeness is hierarchical:

1. Assume that the expert subsystems satisfy their specifications. Using this information, prove that the system reaches a final state.

2. Prove that the expert subsystems satisfy their specifications, and also that they terminate for any possible inputs.

Since a table of one value for each of a set of variables is consistent, state diagram systems that return a set of variable values when they reach a final state are *logically consistent*. The set of variable values may be unsatisfiable, however, given the specifications for the expert system and expert knowledge about the domain.

To show that the output of a state diagram system satisfies a specification for the expert system demonstrate that:

- For each state, if the specifications are satisfied on entering the state, they are also satisfied when leaving the state.
- The specifications are satisfied at the start state. Often the specifications are trivially satisfied at the start state, because the values of output variables are unknown.
- The system always reaches a final state.

Satisfaction Of Specifications: To prove that a specification for a state diagrams is satisfied, one should prove that for any input in the input set of the specification, the state diagram eventually reaches a final state in which the requirements of the specification are satisfied.

Flowcharts

Introduction

Flowcharts are another method for recording expert knowledge and can serve as a model for the knowledge in an expert system.

Use During Development

Flowcharts can be implemented best by using a procedural programming language, i.e., a language that permits:

- Blocks, i.e., sequences of statements used as a single statement.
- Branching statements, e.g., if-then-else or switch statements.
- Loops, e.g., while, do and for loops.
- Function calls, permitting a procedure to call other procedures or itself.

If, however, some procedural knowledge is included in a largely non-procedural knowledge base and the available implementation shell does not permit procedural programming, it may be more convenient to encode the procedural knowledge in rules.

In this case, a flowchart can be represented in rule form by associating a state with each box in the flowchart and by writing rules that describe the transitions between boxes represented by the lines in the flowchart.

Use During VV&E

Completeness, consistency, and satisfaction of specifications for flowcharts are similar to the problems for state diagrams.

If the effect of the flowchart is to set variable values, slot values on objects, or build other data structures, the logical statements represented by these structures can usually be satisfied. The result of the flowchart is logically consistent but not necessarily consistent with the specifications for the expert system or other expert knowledge about the application domain.

Consistency: Flowcharts need not produce consistent output even when:

- The flowchart always reaches an exit box.
- All of the variables that are outputs of the system have a unique value.

If, however, all possible tuples (ordered list of variables) of output variable values are consistent i.e., for any assignment of values to output variables, it is logically possible, and consistent with domain expertise, for the variables to have those variables simultaneously. Then, if for all inputs, the flowchart defines unique values for all the output variables, the flowchart is consistent.

Completeness: A flowchart is logically complete, if no matter what the inputs, the flowchart always reaches an exit box. For this to be true, the one must prove that:

- The computation eventually exits from any loop entered within the flowchart.
- All functions called within the flowchart satisfy their specifications for all inputs and perform their computation in a finite time.

Satisfaction Of Specifications: To show that a flowchart system satisfies its specifications, the basic strategy is to show that if the specifications are satisfied on entry to each box in the flowchart, they are satisfied on exit from the box. Specifications are generally satisfied before the initial box because variables are not yet set to values, but indicating that specifications are satisfied at the start is a necessary part of the proof of specifications.

If box A of the flowchart has just one exit line L going to box B, then A, B, and L represent a sequence of separate computations. To show that this part of the flowchart satisfies the specifications, one should demonstrate that:

- The computations in A and B can always be carried out in only finite time.
- If the specifications are satisfied on entry to A and B, they are satisfied on exit. In proving the specifications for B the user can assume the results of the computations in A, in addition to the specifications that were assumed on entry to A.

If box A of the flowchart performs a test to decide a proposition P, and if A has exits to box B if P is true and box C if P is false, then the user must demonstrate that:

- The specifications are true at the exit box(es) when starting at B with the assumption of the specifications plus P, and that the computation always reaches an exit box in a finite computation.
- The specifications are true at the exit box(es) when starting at C with the assumption of the specifications plus not P, and that the computation always reaches an exit box in a finite computation.

If a flowchart contains a loop, one must demonstrate that, for all inputs satisfying the specifications, the following criteria are met:

- The specs are true on exit from the loop.
- Given the following assumptions at the loop exit:
    - The specifications.
    - The results of computations in the loop.
    - The conditions for exit from the loop.

  The flowchart computation reaches an exit box in finite time and the specifications are true when reaching the exit box.

Functionally Modeled Expert Systems

Introduction

As discussed in the chapter on partitioning without expert knowledge (see Chapter 5), an expert system can be thought of as a *function*. A function maps sets of inputs (information the expert system receives from the user or other external sources) into a set of outputs reflecting actions taken and conclusions inferred by the expert system. Ideally, the function that an expert system represents is that which maps each set of problem inputs into the set of actions and inferences that an expert would make given those inputs. The expert system will be said to *implement* this function, and the function will be said to *model* the expert system with the understanding that an expert system only approximates the behavior of an expert.

Some functions are built from simpler functions with operations such as (function) composition or Cartesian product (operations discussed in more detail below). Sometimes, because of domain knowledge, the expert system should represent a function that is constructed from simpler functions. If that is the case, the structure of the function provides the knowledge engineer with tools for structuring and partitioning an expert system.

More particularly, the operations of Cartesian product and function composition in the category of functions are of particular importance in modeling expert systems. Let E be an expert system such that the output of E involves setting variables O1,...,On such that the values of the O's are independent of each other. Then E implements the Cartesian product of functions fi such that Oi = fi(Ii), where Ii is a subset of the inputs of the entire expert system found by computing the dependency relation (see Chapter 6 on partitioning without expert knowledge) starting with Oi.

If one of the fi is a composition of functions, e.g.

fi = h(g1(Ii), ..., gm(Ii))

then using the same techniques of Chapter 6, one can find subsystems of the original expert system that implement the g's and h can be found.

As discussed in more detail below, if the expert subsystems are complete, consistent and satisfy specifications, *and* if there is consistency and specification satisfaction among independently chosen possible values of Cartesian component subsystems, the entire expert system is complete, consistent and satisfies specifications.

Note that this does *not* mean that completeness, consistency and specifications satisfaction of arbitrary subsets of an expert system imply corresponding results about systems as a whole. The subsets *must* be those that implement functions used to construct the function that models the expert system, *and* certain additional requirements among the outputs of component systems must be met.

Expert knowledge is generally of great benefit in identifying:

- Independent outputs that can be used to decompose an expert system into a product of expert systems.
- Intermediate hypotheses that are functions of the problem inputs but are themselves inputs to a later function that produces some or all of the outputs of the system as a whole.

Following are some examples of composite functions which provide opportunities for structuring and partitioning expert systems.

Use During Development

These strategies often simplify development by replacing a single development task with two or more, which is less than the original task. ***During VV&E***, these strategies likewise replace a single VV&E task with two or more development tasks where the total size is less than the original task.

In each of these cases, the key to whether the partitioning makes these problems smaller is found by counting Hoffman regions. If E is partitioned into E1,...,En, then if:

(H(E1)+...+H(En)) / H(E)

is significantly less than 1, partitioning E into the Ei decreases the size of the development or VV&E problem. Note that usually, *some* rules and variables may be contained in more than one of the Ei.

Cartesian Product Systems: Sometimes an expert system E is required to make more than one decision, e.g., to find values for two different (sets of) variables. In this case, the user can represent the expert system function e of input I as:

e(I) = (e1(I), ..., en(I)).

Using the techniques of chapter 7, the user can find subsystems Ei which implement ei respectively. If H(X) is the number of Hoffman regions in expert system X, then if

(H(E1)+...+H(En)) / H(E)

is significantly less than 1, partitioning E into the Ei decreases the size of the VV&E problem. [Note that *some* rules and variables generally appear in more than one Ei.]

consistency: If each of the Ei is consistent, *and* if the union of consistent sets of output from each of the Ei is consistent, the entire expert system is consistent.

completeness: If each of the Ei is complete, the entire expert system is complete.

specification satisfaction: Generally, proving that specifications are satisfied will involve consideration of the interaction of the outputs of the Ei. However, if a specification is of the form

If C1 and C2 ... and Cn then S (6.1)

then (6.1) is equivalent to the set of specifications

If (AND Ei satisfies Ci) then S.

Final Layer Partitioning: In final layer partitioning, the expert system is partitioned into:

- *The final layer expert system* that consists of all rules and functions that have as their direct outputs conclusions of the knowledge base.
- *Information gathering expert subsystems* that conclude the inputs to the final layer system.

The final layer system contains all rules and functions that produce one or more of the conclusions of the entire expert system. The inputs of the final layer expert system are the inputs to these rules and functions. In KB1, the investment subsystem is the final layer expert system.

For each of the input variables to the final layer expert system, there is an expert system that determines that input to the final level; that expert system can be found using the methods in the chapter on partitioning without expert knowledge. In particular, if the final level input variables are v1,...,vn, let E1,...,En be the expert systems that set these variables.

Those Ei and Ej which overlap greatly, so that:

(H(Ei) + H(Ej)) / H(Ei union Ej) >= 1

should be combined into a single expert system that produces both vi and vj. If, on the other hand,

(H(Ei) + H(Ej)) / H(Ei union Ej)

is significantly less than 1, Ei and Ej should be kept separate. Note that as described in the chapter on partitioning without expert knowledge, clustering of vectors from incidence matrices can be used to determine which of the information gathering subsystems to combine.

Partitioning into a final layer subsystem and information gathering subsystems is particularly useful when there are many rules which compute outputs from the information gathered from the subsystems. PAMEX is an example of such an expert system. In this case, incompleteness or inconsistency in the final layer expert system causes the same error in the entire expert system; furthermore, if there are many rules in the final layer subsystem, such errors are easy to make.

Consistency: The entire expert system is consistent when:

- The final layer expert system is consistent whenever it gets consistent inputs.
- Each of the information gathering subsystems is consistent.
- All unions of consistent output from each of the information gathering subsystems are consistent.

Completeness: If each of the information gathering subsystems is complete and the final layer expert system is complete, then the entire expert system is complete.

Satisfaction Of Specifications: Generally, proving that specifications are satisfied will involve consideration of the interaction of the outputs of the information gathering subsystems.

However, if a specification is of the form:

If C1 and C2 ... and Cn and Cf then S (6.2)

where Ci is a condition on subsystem Ei and Cf is a condition on the final layer,

then (6.2) is equivalent to the set of specifications:

If (AND Ei satisfies Ci)

and the final layer satisfies Cf,

then S is satisfied.

Intermediate Variables: Intermediate variables are variables that are computed or inferred from input variables, and are used to infer or compute conclusions.

Many expert systems can be decomposed into two sequential steps (an expert can often tell the user about such a decomposition):

1. Determine the value of some intermediate variables.

2. Draw conclusions from these intermediate variables.

In addition, an intermediate variable is useful for partitioning only if some of the input variables of the system as a whole are used for computing the intermediate variable.

In function notation, an expert system with an intermediate variable is of the form:

$$e(x1,...,xn) = e(x1,...,xk, y), \text{ where } y = g(xk+1,...,,Xn).$$

Results about completeness, consistency, and specification satisfaction are entirely analogous to those for final level partitioning. However, the role of the final level expert system is that expert system which implements the function:

$$e(x1,...,xk, y)$$

with inputs $x1,...,xk$ and y. This expert system can be found by the method in chapter 5. The single information gathering subsystem is:

$$g(xk+1,...,Xn).$$

Partitioning Of The Function Domain: Let E be an expert system which implements the function $e(I)$, where I is a vector of inputs. Let the domain of I be some domain D, such that D is partitioned into mutually exclusive subsets $\{Di\}$, i.e.,

$$\text{Union}\{Di\} = D$$

$$Di \text{ intersection } Dj = \text{NULL for } i \mathrel{!}= j$$

Let Ei be the expert system that implements the function:

e restricted to Di

Then the following results relate correctness of E to the correctness of the Ei.

Consistency: If each of the Ei is consistent, so is E.

Completeness: If each of the Ei is complete, so is E.

Satisfaction Of Specification: If a specification is satisfied by each Ei, it is satisfied by E.

Examples of domain partitioning occur in decision tree systems. The effect of the decision tree is to partition the entire domain of the expert system into subsets, each of which satisfies the conditions along the path from some leaf node of the decision tree to the root of that tree.

Verifying Knowledge Model Implementations

Overview

Knowledge models are useful for proofs because knowledge models may have certain established properties, such as consistency and completeness, that automatically apply to any system that uses the knowledge model. This means that one can simplify the task of proving something about an expert system by showing that it uses a knowledge model.

However, nothing is free. If one uses a knowledge model to establish the properties of a system, one must show that the system actually uses, i.e. implements, the knowledge model. This requirement is explained below.

Implementation of a Knowledge Model

An expert system implements a knowledge model if:

- The data required by the knowledge model can be identified in the expert system
- The data used in the knowledge model is interpreted by the expert system according to the rules required by the knowledge model.

For example, to show that a rule-based expert system implements a decision tree, it should be shown that:

1. The expert system has rules that fire for each branch of the decision tree
2. The expert system gathers the information needed to select a branch in the decision tree
3. After gathering that information, the expert system selects the branch, i.e. the expert systems the subsystem attached to the branch determined by the just-gathered information.

Proofs Using a Knowledge Model

If a knowledge model is used to establish that an expert system has some property, there are two things that need to be done:

1. Show that for a knowledge base that fits the knowledge model, the desired property is true.
2. Show that the expert system implements the knowledge model. How to do so is the subject of this section.

EXAMPLE

Verifying a System based on Decision Tables

The KB1 demonstration expert system is shown in Figure 4.1. The information in the knowledge base is shown in the following decision tables.

Example Decision Tables:

Investment Decision Table

Risk Tolerance	yes	yes	no	no
Discretionary Income	yes	no	yes	no
Investment	Stocks	Bank Account	Bank Account	Bank Account

Discretionary Income Decision Table

Boat	yes	yes	no	no
Luxury Car	yes	no	yes	no
Discretionary Income	yes	yes	yes	no

Risk Tolerance Decision Table

Lottery Tickets	yes	yes	no	no
Stocks	yes	no	yes	no
Risk Tolerance	yes	yes	yes	no

Analyzing KB1 With These Decision Tables

To illustrate verifying a knowledge base, the knowledge base expressed in the decision tables will be accepted as correct; our current goal is to see that the code implementing the knowledge base contains the information in the decision tables, and only that information.

The following tables shows which rules in Figure 4.1 implement which parts of the decision tables.

Rule	**Table**	**Columns**
1	Investment	1

2	Investment	2 thru 4
3	Risk Tolerance	1 thru 3
4	Risk Tolerance	4
5	Discretionary Income	1 thru 3
6	Discretionary Income	4

To illustrate how this table is interpreted, Row 2 means that Rule 2 implements columns 2 through 4 in the Investment decision table above.

Building the Rule/Decision Table Relation

The Rule/Decision-Table relation was created by inspection, but the information therein is actually the result of simple mathematical reasoning that need not be done in detail, but which must be doable. In particular, it must be shown that whenever the conditions of one of the decision tree columns associated with a rule are true, the rule produces the conclusions(s) of that column of the decision tree. For example, choosing column 2 in the Investment decision table means that:

Risk Tolerance = yes

Discretionary Income = No

This causes Rule 1 to fail and rule 2 to succeed, producing the results of column 2 of the decision table.

It must also be shown that the only way for any rule to fire is to satisfy some column listed for it in the Rule/Decision Table relation. For Rule 2 this follows from the definition of OR, which requires that one of its arguments is true.

The combination of the these two kinds of arguments show that the rules contain the same logical information as the decision tables. However, it is also necessary to show that the expert system actually uses these rules for each branch of the decision tree. This is because it is possible that the inference engine might never fire a rule that would succeed if it were fired. Therefore, to show that an expert system actually implements the decision tree, one must show that the inference engine gathers the necessary information, and fires the right rules.

Verifying and Implemented Expert System Code

To illustrate this process, it will be shown that the implementation code in Clips, shown in Step 3.2 of the Handbook finish this example, implement decision trees similar to those shown above. [The Clips code uses an additional criterion of amount of savings for discretionary income, so the decision trees do not exactly apply to its knowledge base.]

First, we show that Clips gathers the information needed to run the decision tables. Rules a-5c and 3a-3b gather this information. These rules contain only conditions in their if parts that are satisfied when Clips starts, so these rules will be fired, since Clips is forward chaining.

Here we are using properties of Clips described in the User Reference, and are assuming that Clips meets its specifications. This means that we are proving that our knowledge base is correct, assuming Clips meets the specifications we use in the proof. This makes our knowledge base correct conditional on the correctness of Clips, but this is a reasonable compromise in practice. It is much more likely that something new will contain an error than a well-used program like Clips. However, it should be noted that errors have been found in much simpler library programs than Clips, and that in safety-critical systems, the assumptions made about Clips should be verified by testing.

Given that the rules 5a-5c and 3a-3b fire, information needed to put the current problem being run on Clips into some columns of the risk tolerance and discretionary income tables is gathered. This information creates conditions under which 5d and 6 can fire, according to conditions in the rule/decision table relation table. This determines the value of discretionary investment. Similarly, rules 3c and 4 have enough information to be fired by Clips' forward chaining inference engine. This provides the information needed to fire the rules in the investment subsystem (rules 1 and 2). As a result, in all situations, the relevant rules fire. The determination of which rules fire under various decision table conditions is determined by the rule/decision table relation constructed using rules 1, 2, 3c, 4, 5d, and 6. Comparison of these rules with the decision tables, as illustrated above, complete the proof that the Clips system implements the decision tables.

Verifying a System based on State Diagrams

The State Diagram Relation

Following is a table that represents the example state diagram implemented in procedural pseudocode in the VVE Handbook:

State:	**Actions**	**Condition**	**Actions**	**Next State**
start:	_			
start		distance ahead too small	_	distance too small
start		approaching desired exit	_	approaching desired exit
start		clear road ahead	_	clear road
start		default	small delay	start
clear road:	_	current speed <	increment speed,	start

	desired speed	small delay	
clear road	default	small delay	start
distance too small	desired speed and passing possible	pass, small delay	start
distance too small	default	decrease speed, small delay	start
exit: return info to calling program			

The state diagram table has the following meaning:

STATE : ACTIONS names the state and lists actions that are taken when the state is entered. For example, exit : return info to calling program means that on entry to the state exit, return info to calling program is executed. When there is no action to be executed, The dash (-) is used after the state name when there is no action to be executed.

CONDITIONS denotes the entry conditions for a path from the current state. For example, the entry condition for the 2nd. path from the start state is distance too small. Paths from a state are tried in the same order as listed in the table. default may be used for the last path from a state to indicate that that path is always taken if none of the earlier paths are.

ACTIONS denotes the actions taken once the conditions for a path have been satisfied. These are actions that are to be performed when transitioning between a particular pair of states. For example, the actions increment speed, small delay are performed when taking the start path from clear road.

NEXT STATE denotes the next state to go to. For example if the test distance ahead too small is satisfied, the state distance too small is entered.

Showing Code Implements the Diagram Relation

To show that the program implements this state diagram, it is necessary to show that for each row in the state diagram table:

1. There is code that implements the row.
2. That code is executed whenever the state for the row occurs.
3. Nothing else in the program interferes with the code implementing a row.

Condition 1 follows from the fact that there is a branch in the code for each row in the table. A complete verification would identify the computational path for each row. To illustrate the technique, consider the row with the following values:

state = clear road

condition = current speed < desired speed

actions = increment speed, small delay

next state = start

There is a case statement branch corresponding to the clear road state. Both a small delay and setting the next state to start are executed whenever the clear road branch is entered, using the definition of sequential statement execution in procedural languages. Using the definition of if in procedural languages, increment speed is executed whenever current speed < desired speed.

Condition 3 follows from the following two facts:

1. All code in the case statement implements some row in the table

2. The code for each row is executed only when the conditions for that row are satisfied.

Whoops -- A Bug!

In attempting to verify Condition 2, a bug in the code is found. The code for the exit state is never executed. This is because the condition for exiting the while loop succeeds whenever the state becomes exit. Even worse, the only return statement for the code occurs in the erroneously non-executed code for the exit state. As a result, the code shown here could return undefined values to its calling context, propagating errors up through the program in which it is used.

The solution to this bug is to move the return statement to just below the while statement. Were this done, Condition 2 would be satisfied.

The above bug was not planted, but represents a bug in the code that the authors did not catch before writing this section. The fact that the bug was found while trying to carry out a verification proof illustrates that the proof process exposes bugs by causing the developer to examine code greater detail than when he or she merely inspects the code.